Learned Indexes

Sumer Rathinam

Traditional Indexes

- B-Trees: For range requests
- Hash-maps: Single key lookups
- Bloom Filters: Check for record existence

Problem

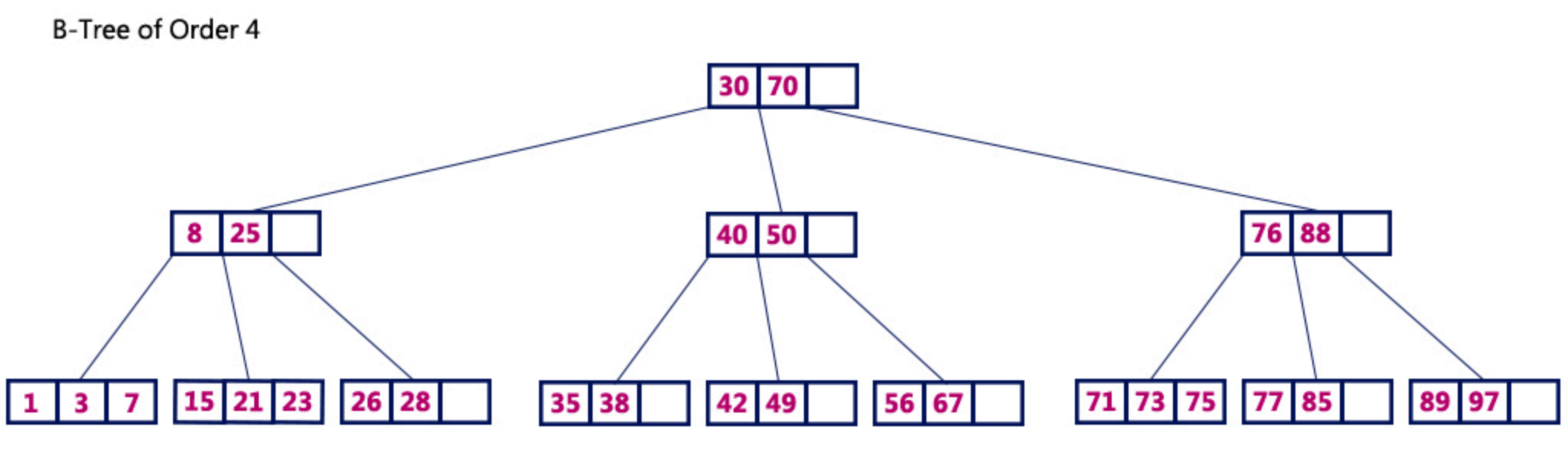
- Traditional indexes are general purpose data structures
- Assume nothing about the data distribution
- Doesn't take advantage of common prevalent patterns in real world data

Example

Goal: Index all integers from 1 to 100M

1, 2, 3, 4, 5, 6, 7, 8, 9, 10 … 100M

B-Tree?



Key Insight

Knowing the exact data distribution allows for instance based optimization

Real World Data

- Real data doesn't follow perfectly known pattern
- Engineering cost to build specialized solution is too high

Machine Learning

- ML can learn a model to reflect data patterns
- Creates specialized index structures
- Low engineering costs
- Cannot provide semantic guarantees
- Traditionally high compute costs

Disclaimers

- Learned indexes are not meant to completely replace existing indexes
- Complement existing work
- Most data structures can be broken down into a learned model and an auxiliary structure
- Continuous functions describing data distribution are used to build efficient data structures and algorithms

3 Key Learned Indexes

- Learned indexes using B-Trees
- Learned indexes using Hash-maps
- Learned indexes using Bloom filters

Range Index

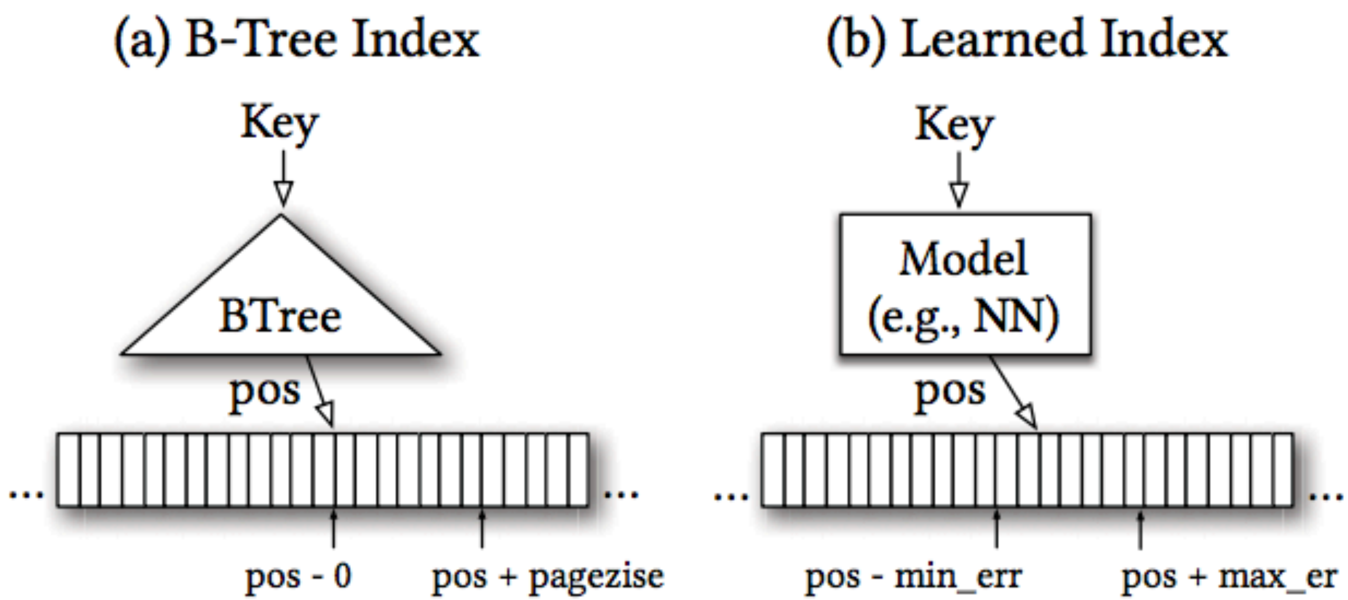


Figure 1: Why B-Trees are models

- Only index every nth key where n is page size
- Min error of 0, max error of the page size
- ML model only needs to provide these error guarantees

ML models

- Have same guarantees as B-Trees
- B-Trees are rebalanced with new data
- ML models retrain to do the same
- Linear regression or neural net are common models that could replace B-Trees

New Challenges

- B-Trees have bounded insert and lookup costs
- Takes advantage of the cache
- Can map keys to pages that are not continuously mapped to memory or disk

* Assumption: we only index an in-memory dense array that is sorted by key

Model Complexity

- Needs to match the same number of operations it takes to traverse B-Tree
- Precision of model needs to be more efficient than a B-Tree

Assumption: B-Tree that indexes 100M records with a page size of 100

With this assumption a model needs to have a better precision gain than1/100 per 400 arithmetic operations

(50 cycles per b-tree page traversal * 8 CPU SIMD operations per cycle)

*This is with all B-Tree pages in cache

CDF Models

- Model that predicts the position of a key inside a sorted array approximates the cumulative distribution function

$$p = F\,(\text{Key}) * N$$

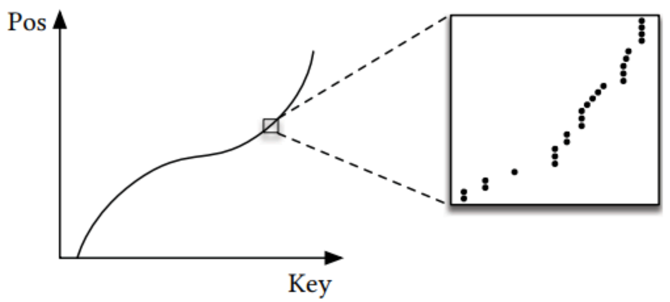


Figure 2: Indexes as CDFs

- p is the position estimate
- F (Key) is the estimated CDF for the data to estimate the likelihood to observe a key smaller or equal to the look-up key P(X ≤ Key)
- N is the total number of keys

Key Takeaways

- B-Tree learns the distribution by creating a regression tree
- ML model can do the same by minimizing the squared error of a linear function
- CDF will play a key role in optimizing other types of index structures

Naïve Learned Index

- Used 200M web server log records
- Built secondary index over the timestamps
- Trained a two-layer fully connected neural network with 32 neurons per layer
- Timestamps are input features
- Positions in sorted array are the labels
- Took 80,000 nano-seconds to execute
- B-Tree took 300 nano-seconds

Recursive Model Index

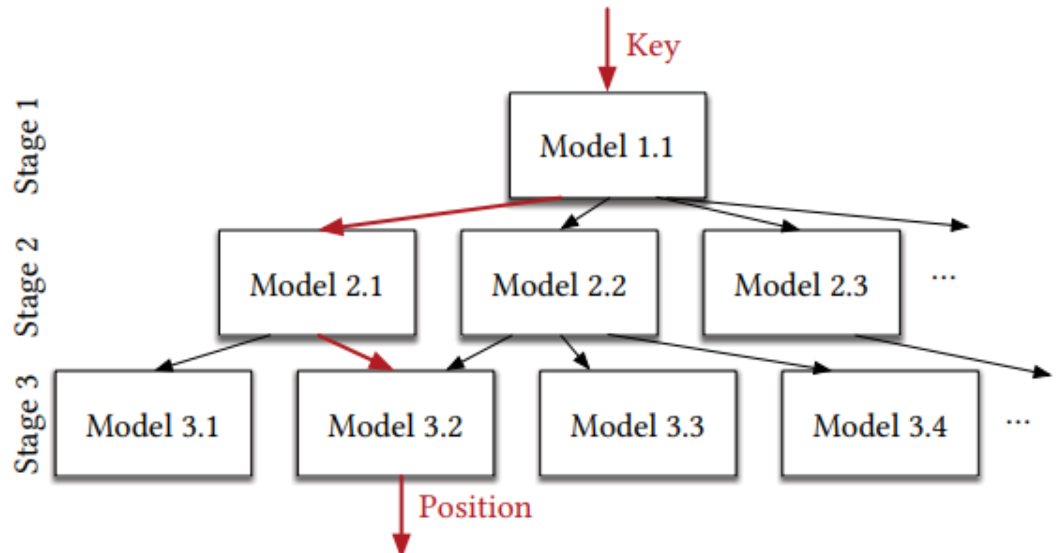


Figure 3: Staged models

- Takes key as input
- Predicts position with certain error
- Selects another model based on error of prediction
- Final stage gives position

Hybrid Indexes

- Recursive model allows for a mixture of models depending on the stage
- Top layers are more likely to use small Neural Nets so they can learn a wide range of data
- Bottom layers can use thousands of simple linear regression models as they are inexpensive in space and execution time
- Paper replaces NN models with B-Trees if absolute min-/max-error is above a predefined threshold

* Hybrid indexes bind the worst case performance of learned indexes to the performance of B-Trees.

Results

		Map Data			Web Data			Log-Normal Data		
Type	**Config**	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)	Size (MB)	Lookup (ns)	Model (ns)
Btree	page size: 32	52.45 (4.00x)	274 (0.97x)	198 (72.3%)	51.93 (4.00x)	276 (0.94x)	201 (72.7%)	49.83 (4.00x)	274 (0.96x)	198 (72.1%)
	page size: 64	26.23 (2.00x)	277 (0.96x)	172 (62.0%)	25.97 (2.00x)	274 (0.95x)	171 (62.4%)	24.92 (2.00x)	274 (0.96x)	169 (61.7%)
	page size: 128	13.11 (1.00x)	265 (1.00x)	134 (50.8%)	12.98 (1.00x)	260 (1.00x)	132 (50.8%)	12.46 (1.00x)	263 (1.00x)	131 (50.0%)
	page size: 256	6.56 (0.50x)	267 (0.99x)	114 (42.7%)	6.49 (0.50x)	266 (0.98x)	114 (42.9%)	6.23 (0.50x)	271 (0.97x)	117 (43.2%)
	page size: 512	3.28 (0.25x)	286 (0.93x)	101 (35.3%)	3.25 (0.25x)	291 (0.89x)	100 (34.3%)	3.11 (0.25x)	293 (0.90x)	101 (34.5%)
Learned Index	2nd stage models: 10k	0.15 (0.01x)	98 (2.70x)	31 (31.6%)	0.15 (0.01x)	222 (1.17x)	29 (13.1%)	0.15 (0.01x)	178 (1.47x)	26 (14.6%)
	2nd stage models: 50k	0.76 (0.06x)	85 (3.11x)	39 (45.9%)	0.76 (0.06x)	162 (1.60x)	36 (22.2%)	0.76 (0.06x)	162 (1.62x)	35 (21.6%)
	2nd stage models: 100k	1.53 (0.12x)	82 (3.21x)	41 (50.2%)	1.53 (0.12x)	144 (1.81x)	39 (26.9%)	1.53 (0.12x)	152 (1.73x)	36 (23.7%)
	2nd stage models: 200k	3.05 (0.23x)	86 (3.08x)	50 (58.1%)	3.05 (0.24x)	126 (2.07x)	41 (32.5%)	3.05 (0.24x)	146 (1.79x)	40 (27.6%)

Figure 4: Learned Index vs B-Tree

- Learned index dominates B-Tree
- Most configurations 1.5 - 3 times faster
- Up to 2 orders of magnitude smaller in size

Indexing Strings

- Tokenize input string into input vector
- Treated the same as real valued keys except with a vector instead of single value
- Linear models scale the number of multiplications and additions linearly with regards to input length

Results For Strings

	Config	Size(MB)	Lookup (ns)	Model (ns)
Btree	page size: 32	13.11 (4.00x)	1247 (1.03x)	643 (52%)
	page size: 64	6.56 (2.00x)	1280 (1.01x)	500 (39%)
	page size: 128	3.28 (1.00x)	1288 (1.00x)	377 (29%)
	page size: 256	1.64 (0.50x)	1398 (0.92x)	330 (24%)
Learned Index	1 hidden layer	1.22 (0.37x)	1605 (0.80x)	503 (31%)
	2 hidden layers	2.26 (0.69x)	1660 (0.78x)	598 (36%)
Hybrid Index	t=128, 1 hidden layer	1.67 (0.51x)	1397 (0.92x)	472 (34%)
	t=128, 2 hidden layers	2.33 (0.71x)	1620 (0.80x)	591 (36%)
	t= 64, 1 hidden layer	2.50 (0.76x)	1220 (1.06x)	440 (36%)
	t= 64, 2 hidden layers	2.79 (0.85x)	1447 (0.89x)	556 (38%)
Learned QS	1 hidden layer	1.22 (0.37x)	1155 (1.12x)	496 (43%)

Figure 6: String data: Learned Index vs B-Tree

- 10M non continuous document IDs of a large web index
- Learned QS is a non hybrid recursive model index using quaternary search
- Best performance for strings, while normal learned index did not perform as well

Point Index

- Hash-maps traditionally used
- Key is to prevent too many conflicts
- Example:
  - 100M records
  - Hash-map size of 100M
  - Uniformly random keys
  - Leads to 33% or 33M conflicts

*Machine learning models can reduce conflict

ML Models

- Using learned models as a hash-function already exists
- Existing solutions don't take advantage of underlying data distribution
- Machine learning models can provide a more customized solution

Comparison

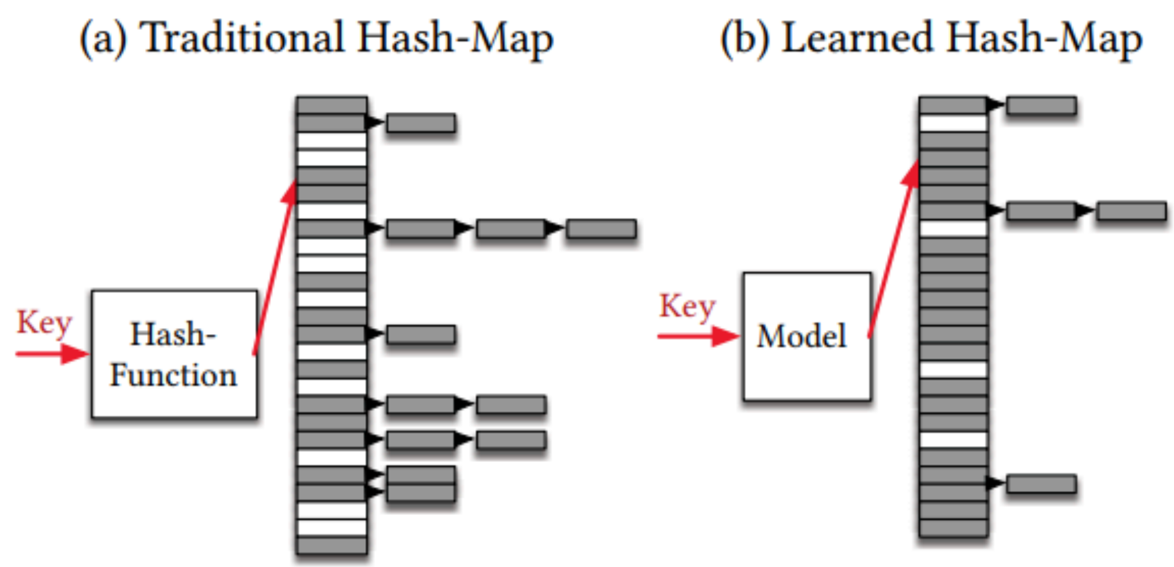


Figure 7: Traditional Hash-map vs Learned Hash-map

- $H(K) = F(K) * M$, M is the size of the hash-map
- Scales the CDF by targeted size of M
- If we perfectly learn the CDF of keys, no conflicts would occur
- Uses the same recursive model index as before

Hash Model

- Tradeoff between size of index and performance
- Benefits of learned model depend on
  - How accurately the model represents the CDF
  - Hash map architecture

Example:

- With small keys and little to no values, traditional hash functions will perform well
- With larger payloads learned models will perform better

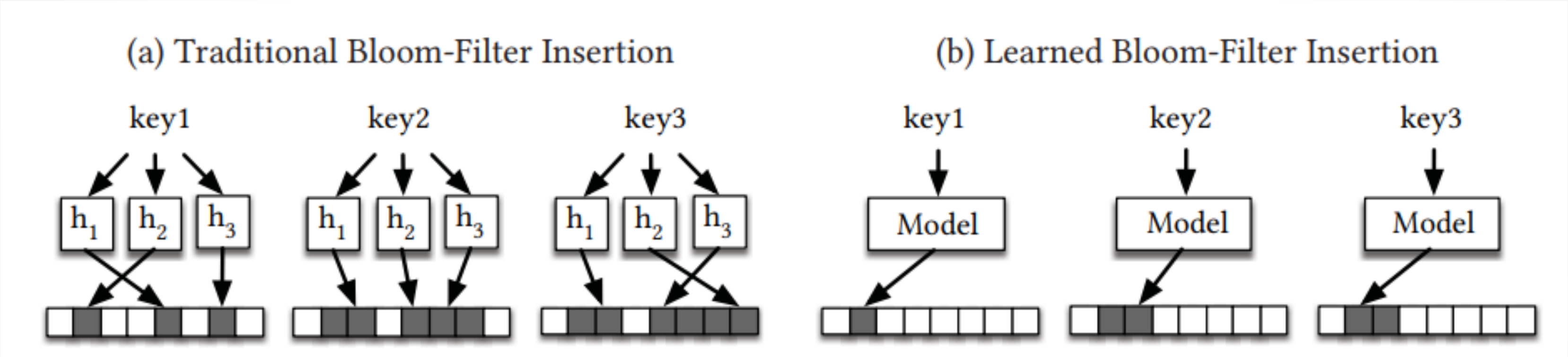
Results

	% Conflicts Hash Map	% Conflicts Model	Reduction
Map Data	35.3%	07.9%	77.5%
Web Data	35.3%	24.7%	30.0%
Log Normal	35.4%	25.9%	26.7%

Figure 8: Reduction of Conflicts

- Used the same 3 sets of data from b-tree evaluation
- 2 stage recursive model index used
- 100k models on the second stage

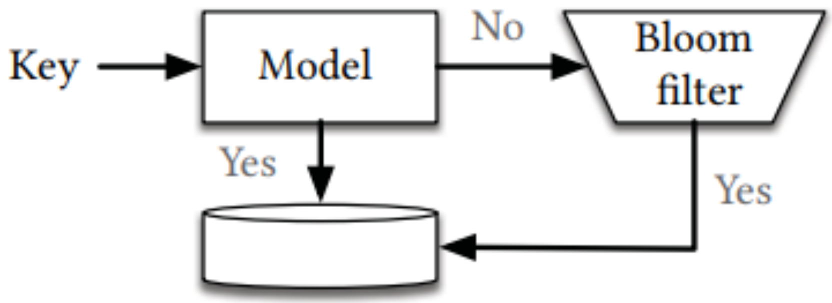
Existence Index



- Traditional bloom filters are space efficient, but still can occupy a lot of memory
- False negative rate of 0
- Specific false positive rate
- Learned model can achieve these requirements

Existence Index Model

(c) Bloom filters as a classification problem



- Learn a model f that predicts whether query x is a key or non key
- Use Recurrent NN or Convolutional NN to do this
- Will need an overflow bloom filter to keep false negative rate at 0
- Still has a certain false positive rate

Results

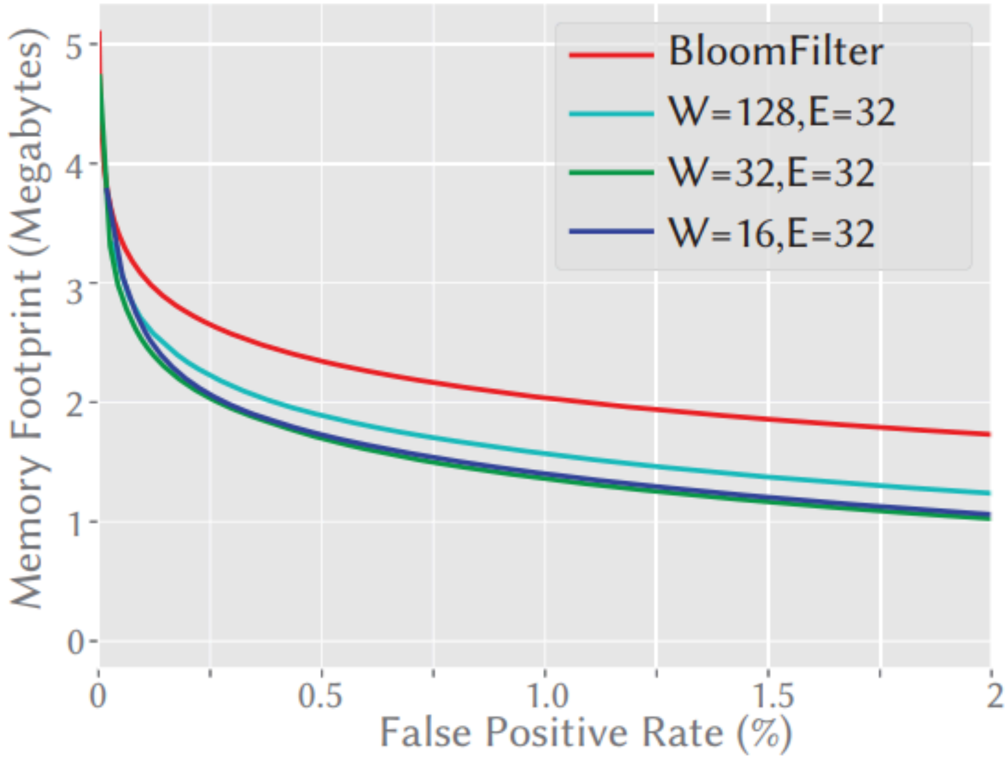


Figure 10: Learned Bloom filter improves memory footprint at a wide range of FPRs. (Here *W* is the RNN width and *E* is the embedding size for each character.)

Future Work

- Using other ML models i.e. not just linear models and NN
- Multidimensional indexes i.e. position of all records filtered by any combination of attributes
- Beyond indexing: learned algorithms
  - Learning the CDF model could speed up sorting and joins, not just indexes
- GPU/TPU improvements and speedups

Overall Thoughts

- Does a great job of putting complex concepts into simple terms
- The mapping between traditional indexes and learned models is great
- Experiments were well thought out and covered worst cases
- Could've talked more on how these new findings will impact the industry
- How can we get learned indexes into some sort of commercial system